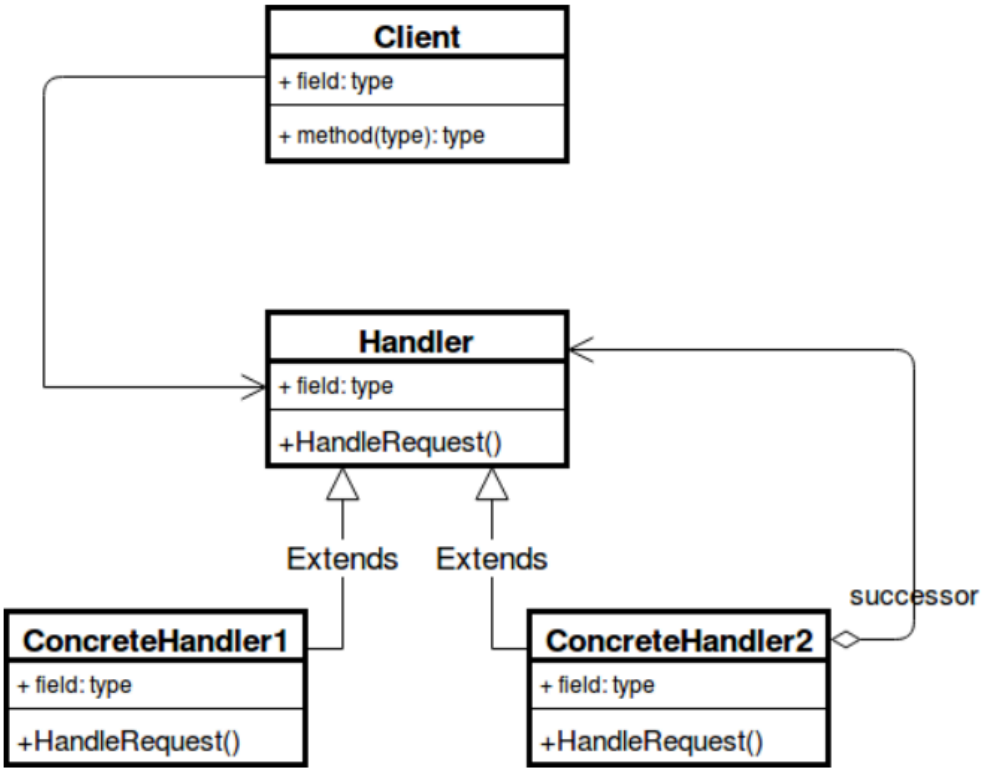
책임 연쇄(Chain of Responsibility) / 커맨드(Command)

책임 연쇄(Chain of Responsibility)

- 책임이란 무언가를 처리하는 기능(클래스)
- 여러 개의 책임들을 동적으로 연결해서 순차적으로 실행하는 패턴
- 기능을 클래스 별로 분리하여 구현하도록 유도하므로 클래스의 코드가 최적화됨

---

1) UML



- `Handler` : 요청을 수신하고 처리 객체들의 집합을 정의하는 인터페이스
- `ConcreteHandler` : 요청을 처리하는 실제 처리 객체
  - 핸들러에 대한 필드를 내부에 가지고 있으며, 메서드를 통해 다음 핸들러를 체인으로 연결한다.

    - 자신이 처리할 수 없는 요구가 나오면 다음 체인의 핸들러에게 요청을 전달한다.
    - `ConcreteHandler1` - `ConcreteHandler2` - `ConcreteHandler3` ...이런 식으로 체인 형식으로 구성된다.
- **`Client`** : 요청을 `Handler`에 전달

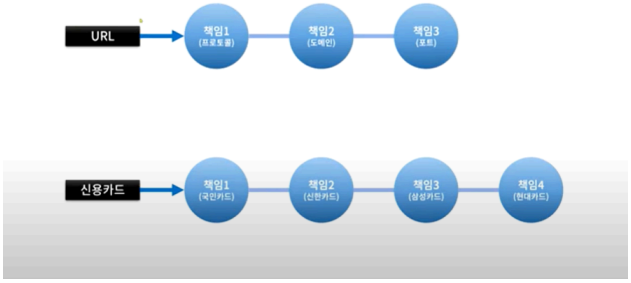
※ 여기서 **`ConcreteHandler3`**를 새로 추가해야 한다고 가정해보자. 이 경우 기존 클래스들을 전혀 수정하지 않고 새로운 클래스를 추가하는 것만으로 기능을 확장할 수 있다. 이는 **최적화된 설계 구조**가 가진 장점 중 하나로, 유지보수성과 확장성을 동시에 확보할 수 있는 이상적인 사례다.

> 이는 책임 연쇄 패턴의 핵심적인 장점을 보여주는 중요한 예시이다. 기존 시스템의 변경 없이 새로운 기능을 추가할 수 있다는 것은 개방-폐쇄 원칙(OCP)을 잘 준수하고 있음을 의미한다.

체인의 각 구성 요소는 자신의 역할에만 집중하여 동작하므로, 새로운 요청을 처리할 객체를 추가하는 작업이 간단하고 명료하다. 더불어, 클라이언트 코드를 수정하지 않고도 핸들러를 체인에 동적으로 추가하거나, 처리 순서를 변경하거나, 특정 핸들러를 삭제하는 등의 작업이 가능해져 설계 구조가 훨씬 유연해진다.

---

2) 책임 연쇄 패턴 적용 예시



1. 첫 번째 예시 : `http://www.youtube.com:1007`과 같은 URL 문자열을 입력받는 경우, 책임 연쇄를 통해 각 구성 요소를 순차적으로 처리한다.
    - **책임 1**은 URL의 프로토콜(`http`)을 식별하고 처리한다.
    - **책임 2**는 URL의 도메인(`www.youtube.com`)을 추출하고 처리한다.
    - **책임 3**은 URL의 포트 번호(`1007`)를 확인하고 처리한다.

    최종적으로 다음과 같은 결과를 출력한다.
    - **PROTOCOL**: `http`
    - **DOMAIN**: `www.youtube.com`
    - **PORT**: `1007`

2. 두 번째 예시 : 신용카드 정보를 입력받은 후, 처리 책임을 단계별로 위임하여 카드 발급사를 식별하고, 해당 발급사에 맞는 처리를 수행한다. 이 과정은 책임 1부터 책임 4까지 순차적으로 진행되며, 각 단계는 자신이 처리 가능한 카드 발급사인지 확인 후, 해당 사항이 없으면 다음 단계로 처리를 위임한다.

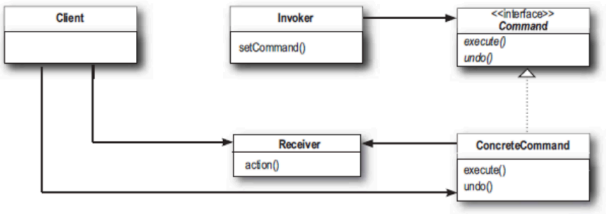
   예를 들어, 입력된 신용카드가 신한카드에 해당하는 경우, 책임 1에서는 이를 처리할 수 없으므로 다음 단계로 책임을 넘긴다. 책임 2에서 신한카드 발급사임을 확인하고 적절한 처리를 수행하며, 이후 단계(책임 3, 책임 4)는 실행되지 않는다

커맨드(Command)

- 하나의 명령(기능)을 객체화한 패턴
- 객체는 전달할 수 있고 보관할 수 있음, 즉 명령(기능)을 전달하고 보관할 수 있게 됨
- 배치 실행, Undo/Redo, 우선순위가 높은 명령을 먼저 실행하기 등이 가능해짐

---

1) UML



- `client` : `ConcreteCommand`를 생성하고 `Receiver`를 설정

- `Receiver` : 요구 사항을 수행하기 위해 어떤 일을 처리해야 하는지 알고 있는 객체
  - `ConcreteCommand`에서 `Receiver`를 가지고 있으며 `ConcreteCommand`는 명령에 따라 `Receiver`의 기능을 수행한다.

- `ConcreteCommand` : 특정 행동과 `Receiver` 사이를 연결해 준다.
  - `Invoker`에서 excute(), undo() 등 호출을 통해 요청이 들어오면 `ConcreteCommand`에 있는 `Receiver` 객체의 메서드를 호출해 작업을 처리하게 된다.

- `Invoker` : 명령이 들어 있으며 excute(), undo() 등 메서드를 호출함으로써 `Command` 객체에게 특정 작업을 수행해달라는 요구를 하게 된다.

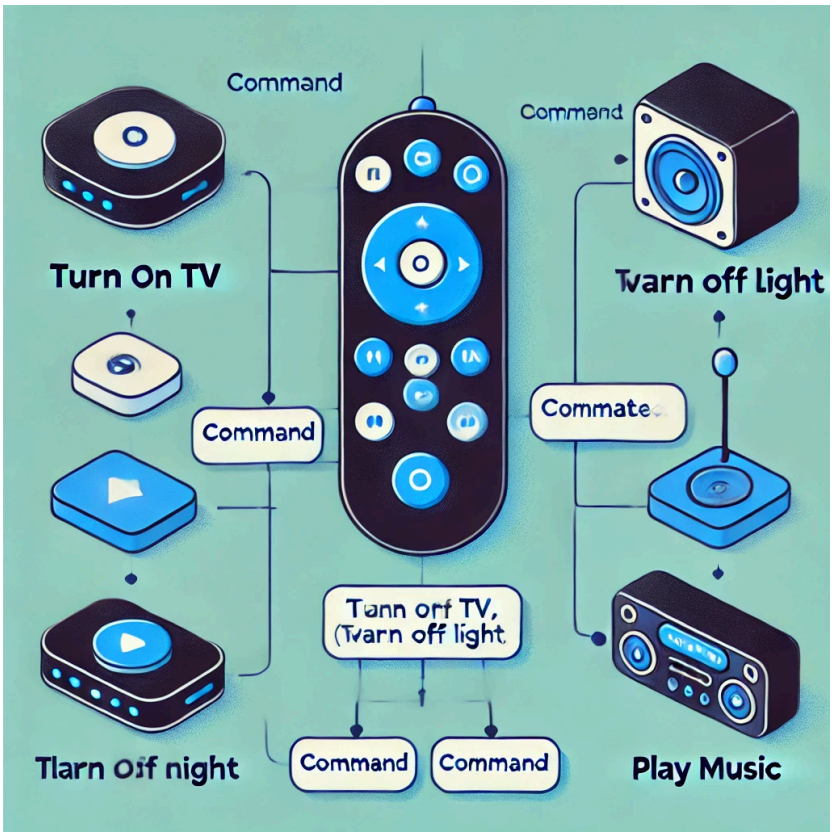
- `Command` : 모든 `Command` 객체에서 구현해야 하는 인터페이스이며 `Receiver`에 특정 작업을 처리하라는 `Invoker`의 지시를 전달한다.

※ 쉽게 말해서, 커맨드 패턴은 '무엇을 실행해야 하는지'를 독립적인 객체로 만들어 요청(명령)을 객체로 **캡슐화**하여, 실행할 작업과 호출자를 분리하는 디자인 패턴이다.

> 작업을 처리할 특정 객체를 지정하지 않은 채 처리하고자 하는 작업을 객체 집단에 던지는 책임 체인 패턴과 달리, 커맨드 패턴은 처리를 담당할 객체를 직접 지정한 후 그 객체를 Command란 처리 박스에 요청하여 처리하는 패턴이다.

---

2) 커맨드 패턴 적용 예시



- 위 그림은 **커맨드 패턴**의 동작 방식을 간단히 설명한다.

- 스마트 리모컨의 각 버튼은 **명령 객체**를 통해 TV, 조명, 음악 시스템과 같은 장치(수신자)로 요청을 전달한다.

- 이 패턴을 통해 명령은 독립적으로 관리되며, 실행 취소(Undo)나 명령 큐 같은 기능도 쉽게 구현할 수 있다.

예시에서 중요한 점은 버튼)이 어떤 작업이 실행되는지 알 필요가 없으며, 모든 작업을 명령 객체(Command Object)에 위임한다는 것이다. 이러한 구조는 SOLID 객체지향 5대 원칙 중 다음과 같은 원칙들을 잘 반영한다.

1. **단일 책임 원칙(SRP)**:

   버튼은 단순히 요청을 전달하는 역할만 수행하며, 실제 작업의 세부 구현은 명령 객체와 수신자(Receiver)가 담당한다. 이를 통해 각 클래스가 하나의 책임만 가지도록 설계된다.

2. **개방-폐쇄 원칙(OCP)**:

   새로운 명령을 추가할 때 기존 코드(버튼이나 기존 명령 객체)를 수정하지 않고, 새로운 명령 객체만 생성하여 연결할 수 있다. 이는 시스템의 확장성을 높인다.